Lecture 8

- Recap of classes
- Inheritance
- Polymorphism
- Interfaces
- Overview of projects
Classes – a recap

Classes contain

- variables *(member data or member variables)*
- functions *(member functions or methods)*

```cpp
class Point2D
{
public:
    double x, y; // <-- member variables
    double DistanceToOrigin() // <-- a member function
    {
        return std::sqrt(x*x+y*y);
    }
};
```
A class defines a new type (like `int`, `std::vector<double>`, etc.). We create objects of this type in the usual way:

```cpp
Point2D a, b;
```

We access the public variables and functions in a class using the dot `. operator, or the `->` operator for pointers:

```cpp
Point2D a, b;
a.x = 3; a.y = 4; // use . for objects

Point2D *pb = &b; // pb is a pointer to b
pb->x = 3; pb->y = 4; // use -> for pointers to objects
```
The variables and functions in a class can be

- **public**: accessible from outside the class, as in previous slide
- **private**: accessible only to functions within that class
- **protected**: accessible only to functions in that class or in derived classes

```cpp
class Point2D
{
private:
    double x, y; // <-- private member variables
public:
    double DistanceToOrigin()
    {
        return std::sqrt(x*x+y*y); // only accessible within a
                                  // member function such as this
    }
};
```
Classes – a recap

The *constructor* is a member function

- with the same name as the class
- with no return value

that is executed when the class is created:

```cpp
class Point2D
{
public:
    Point2D(double x_, double y_) // a constructor
    {
        x = x_; y = y_; 
    }
private:
    double x, y;
};

Point2D a(3, 4); // when creating vars, use constructor arg.s
```
Inheritance and polymorphism: Motivation

• Suppose we want to write a function to calculate

\[ \int_a^b f(x) \, dx \]

for arbitrary \(a, b\) and \(f(x)\).

• We can pass \(a\) and \(b\) into the function as `double` parameters, but how can we pass the integrand \(f\) as a parameter?

```cpp
double Integrate(double a, double b, ???function f???)
{
    // ... numerical integration routine here, calls f(x)
}
```

• One way of doing this is through class inheritance.
Inheritance

- C++ allows the creation of new classes that inherit from existing classes.
- We say the new class is derived from an existing base class.
- The derived class inherits (contains) all the member variables and member functions from the base class, and can
  - Define new member variables and functions
  - Redefine member functions in the base class
- Inheritance often defines an ‘is a’ relationship:
  - Classes Triangle and Rectangle might both derive from a base class Polygon
- Inheritance allows polymorphism:
  - a base class reference/pointer can point to an object of any type derived from that base class
  - calling (virtual) member functions from this base class pointer/reference will execute the (overridden) member function in the derived class
Inheritance

We define a base class Polygon:

class Polygon
{
public:
    Polygon(double w, double h) {width = w, height = h};
protected:  // can be accessed in this and derived classes only
    double width, height;
};

and a class Triangle which inherits from Polygon

// This syntax defines a class Triangle that inherits from Polygon
class Triangle : public Polygon
{
public:
    Triangle(double w, double h) : Polygon(w, h) {}
    double Area() { return 0.5*width*height; }
};
Inheritance

We can use the derived `Triangle` class as we use any other class:

```cpp
// Triangle inherits the constructor from Polygon
Triangle t(3.0, 4.0);
// Use the function defined in Triangle
std::cout << t.Area() << std::endl;
```

We can assign a `Polygon` pointer or reference to the `Triangle`...

```cpp
Polygon *pP = &t; // ok, pointer contains address of t
pP->Area(); // error! Area is not a member function of Polygon
```

...but we can only access members of `Polygon` through this pointer:

```cpp
void DoSomething(Polygon &p)
{
    p.Area(); // error! Area is not a member function of Polygon
}

DoSomething(t); // ok to call with Triangle object t
```
Virtual functions

How can we call `Triangle::Area()` via a base class (`Polygon`) pointer or reference?

If we define an `Area` function in the class `Polygon`...

```cpp
class Polygon
{
public:
    Polygon(double w, double h) {width = w, height = h};
    double Area() {return -1;}
protected: // can be accessed in this and derived classes only
    double width, height;
};

...then the code compiles, but calls `Polygon::Area()` instead:
```

```cpp
Triangle t(3.0, 4.0);
Polygon *pP = &t;
t.Area(); // This calls `Triangle::Area()
pP->Area(); // This compiles, but calls `Polygon::Area()`
```
Virtual functions

How can we call Triangle::Area() via a base class (Polygon) pointer or reference?

The solution is to define the Area function in Polygon as virtual:

```cpp
virtual double Area() { return -1; }  // virtual function
```

Triangle t(3.0, 4.0);
Polygon *pP = &t;
t.Area();  // This calls Triangle::Area()
pP->Area();  // This calls Triangle::Area()

• We say that the Triangle::Area() function overrides the Polygon::Area() function.

• It is good practice to use the virtual keyword in the definition of Triangle::Area() too.
Virtual functions

How can we call `Triangle::Area()` via a base class (`Polygon`) pointer or reference?

We could define the `Polygon::Area` function to be pure virtual, with no implementation in `Polygon::Area`:

```cpp
virtual double Area() = 0; // pure virtual function
```

Pure virtual functions act exactly as virtual functions, except that a class which has a pure virtual method cannot be instantiated:

```cpp
Polygon p(1.0, 2.0);       // Error! Polygon::Area() is pure virtual
Triangle t(1.0, 2.0);      // ok, since Triangle overrides Area
Polygon *pP = &t;         // ok: we are allowed pointers to Polygon
```
class Base
{
public:
    void X() {std::cout << "base::X" << std::endl;}
    virtual void Y() {std::cout << "base::Y" << std::endl;}
    virtual void Z() = 0;
};

class Derived : public Base
{
public:
    void X() {std::cout << "derived::X" << std::endl;}
    virtual void Y() {std::cout << "derived::Y" << std::endl;}
    virtual void Z() {std::cout << "derived::Z" << std::endl;}
};

Base b;
Derived d;
d.X(); d.Y(); d.Z();
Base *pd = &d;
pd->X(); pd->Y(); pd->Z();
Summary of virtual functions

class Base
{
public:
    void X() {std::cout << "base::X" << std::endl;}
    virtual void Y() {std::cout << "base::Y" << std::endl;}
    virtual void Z() = 0;
};

class Derived : public Base
{
public:
    void X() {std::cout << "derived::X" << std::endl;}
    virtual void Y() {std::cout << "derived::Y" << std::endl;}
    virtual void Z() {std::cout << "derived::Z" << std::endl;}
};

Base b; // error! pure virtual Z() prevents object of type Base
Derived d;
d.X(); d.Y(); d.Z(); // derived::X, derived::Y, derived::Z
Base *pd = &d;
pd->X(); pd->Y(); pd->Z(); // base::X, derived::Y, derived::Z
Inheritance – a recap

What if a function in the derived class has the same signature (name, parameters, return value) as one in the base class?

We can define functions in the base class as

1. Normal functions `int Function() { return 5; }
2. Virtual functions `virtual int Function() { return 5; }
3. Pure virtual functions `virtual int Function() = 0`
   - with a (non-pure) virtual implementation in the derived class
   - prevents creation of a base class object

We can access these functions through

1. A base class object `Base b; b.Function()`
2. A derived class object `Derived d; d.Function()`
3. A base class pointer or reference to a derived class object:

   ```
   Derived d;
   Base *pb = &d;
   pb->Function();
   ```
Inheritance – a recap

What if a function in the derived class has the same signature (name, parameters, return value) as one in the base class?

Whether the base or derived member function is called depends on

1. the type of function in the base class, and
2. the object with which we are accessing it

<table>
<thead>
<tr>
<th></th>
<th>function</th>
<th>virtual</th>
<th>pure virtual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base object</td>
<td>Base</td>
<td>Base</td>
<td>Error!</td>
</tr>
<tr>
<td>Derived object</td>
<td>Derived</td>
<td>Derived</td>
<td>Derived</td>
</tr>
<tr>
<td>Base pointer/ref. to derived</td>
<td>Base</td>
<td>Derived</td>
<td>Derived</td>
</tr>
</tbody>
</table>
Interfaces

- A class with pure virtual functions is called an Abstract Base Class (or ABC).
- Abstract base classes can be used to define a common interface for (or way of communicating with) a range of derived classes.

Example

- We are writing a function to calculate

  \[ \int_{a}^{b} f(x) \, dx \]

  for arbitrary \( a, b \) and \( f(x) \).
- We can pass \( a \) and \( b \) into the function as double parameters, but how can we pass the integrand \( f \) as a parameter?
The interface for an integrand is that, given a (double) parameter \( x \), it should evaluate to some (double) value.

We can represent this in an abstract `Integrand` class:

```cpp
class Integrand {
public:
    virtual double Evaluate(double x) = 0;
};
```

Now we can write our function to integrate an arbitrary integrand:

```cpp
double Integral(double a, double b, Integrand &i) {
    // Trapezium rule with one trapezium
    return (b-a)*0.5*(i.Evaluate(a)+i.Evaluate(b));
}
```
Interfaces

For any particular function we choose, we can create a class derived from the *Integrand* class:

```cpp
class SinIntegrand : public Integrand
{
    public:
        virtual double Evaluate(double x)
        {
            return std::sin(x); // in the <cmath> library
        }
};
```

We can then call our *Integral* function, passing an object of the derived class type

```cpp
SinIntegrand si;
std::cout << Integral(0, 0.5, si) << std::endl;
```

The calls to *Evaluate* in the *Integral* function will call the overridden function in our *SinIntegrand* class.
class Integrand // base class definition
{
public:
    virtual double Evaluate(double x) = 0; // with pure virtual method
}; // end of base class definition

class SinIntegrand : public Integrand // derived class definition
{
public:
    virtual double Evaluate(double x) // implementation of method
    {
        return std::sin(x);
    }
}; // end of derived class definition

double Integral(double a, double b, Integrand &i)
{
    // i is a reference to base class
    return (b-a)*0.5*(i.Evaluate(a)+i.Evaluate(b));
}

int main()
{
    SinIntegrand si; // Call integral with object of derived class
    std::cout << Integral(0, 0.5, si) << std::endl;
}
Project 1

- Set today (see website for projects booklet)
- Due in 3pm, Friday 15th November 2019
- Worth 40% of the course
- Marked like a dissertation, not like short courseworks
- Don’t spend too much time on projects!
  - Median student might complete most but not all of a project.
  - About 40 hours work outside of labs
    - 15 credit unit × 10 study hours per credit = 150 hours total
    - Subtract 10 hours lectures, 22 hours labs, 18 hours lecture study
    - Leaves 1 hour per % of assessed work
- 15 pages of writing MAXIMUM
- Quality over quantity
1. ODE Boundary Value Problems

- Applied mathematics / fluid dynamics
- Inviscid fluid flow over a wedge leads to a boundary layer
- The thickness of this is determined by the Falkner-Skan equation

\[
\begin{align*}
    f''' &+ ff'' + \beta (1 - f'^2) = 0, \\
    f(0) & = f'(0) = 0, \\
    f' & \rightarrow 1 \quad \text{as} \quad \eta \rightarrow \infty.
\end{align*}
\]

- This is a nonlinear boundary value problem
- Project extends techniques for ODE initial value problems to solve BVPs using the ‘shooting’ method
2. Volatility

- Financial mathematics
- Recommended project for MSc QFFE
- Extraction of historical volatility from share and option prices
- Evaluation of implied volatility of an option from Black-Scholes
- Requires knowledge of finance and financial mathematics
3. Conjugate Gradient method

- Numerical linear algebra
- Conjugate gradients is an iterative algorithm for solving
  \[Ax = b\]
- Can be use on large matrices \((10^6 \times 10^6\) or more)
- Investigation of:
  - the influence of the matrix on convergence rates
  - speedup due to sparse matrix representations
- (Variants of) CG used widely in high performance scientific computing
4. Sorting

- Computer science
- Investigation of a few common sorting algorithms
  - bubble sort
  - quicksort
  - heapsort
- Investigation of run-time for these algorithms
- Use of sorted data structures for an optimised implementation of Conway's *game of life*